CHAPTER 1
Foundations of Data Analysis and Python

Introduction

In today's data-rich landscape, data is much more than a collection of numbers or facts, it's a powerful resource that can influence decision-making, policy formation, product development, and scientific discovery. To turn these raw inputs into meaningful insights, we rely on statistics, the discipline dedicated to collecting, organizing, summarizing, and interpreting data. Statistics not only helps us understand patterns and relationships but also guides us in making evidence-based decisions with confidence. This chapter examines fundamental concepts at the heart of data analysis. We'll explore what data is and why it matters, distinguish between various types of data and their levels of measurement, and consider how data can be categorized as univariate, bivariate, or multivariate. We'll also highlight different data sources, clarify the roles of populations and samples, and introduce crucial data preparation tasks including cleaning, wrangling, and manipulation to ensure data quality and integrity.

For example, consider you have records of customer purchases at an online store everything from product categories and prices to transaction dates and customer demographics. Applying statistical principles and effective data preparation techniques to this information can reveal purchasing patterns, highlight which product lines drive the most revenue, and suggest targeted promotions that improve the shopping experience.

Structure

In this chapter, we will discuss the following topics:

- Environment setup
- Software installation
- Basic overview of technology
- Statistics, data, and its importance
- Types of data
- Levels of measurement
- Univariate, bivariate, and multivariate data
- Data sources, methods, population, and samples
- Data preparation tasks
- Wrangling and manipulation

Objectives

By the end of this chapter, readers will learn the basics of statistics and data, such as, what they are, why they are important, how they vary in type and application, and the basic data collection and manipulation techniques. Moreover, this chapter explains different level of measurements, data analysis techniques, its source, collection methods, their quality and cleaning. You will also learn how to work with data using Python, a powerful and popular programming language that offers many tools and

libraries for data analysis.

Environment setup

To set up the environment and to run the sample code for statistics and data analysis in Python, the three options are as follows:

- Download and install Python from **https://www.python.org/downloads/**. Other packages need to be installed explicitly on top of Python. Then, use any **integrated development environment (IDE)** like visual studio code to execute Python code.
- You can also use Anaconda, a Python distribution designed for large-scale data processing, predictive analytics, and scientific computing. The Anaconda distribution is the easiest way to code in Python. It works on Linux, Windows, and Mac OS X. It can be downloaded from **https://www.anaconda.com/distribution/**.
- You can also use cloud services, which is the easiest of all options but requires internet connectivity to use. Cloud providers like Microsoft Azure Notebooks, GitHub Code Spaces and Google Collaboratory are very popular. Following are a few links:
    - **Microsoft Azure Notebooks**: **https://notebooks.azure.com/**
    - **GitHub Codespaces**: Create a GitHub account from **https://github.com/join** then, once logged in, create a repository from **https://github.com/new**. Once the repository is created, open the repository in the codespace by using the following instructions: **https://docs.github.com/en/codespaces/developing-in-codespaces/creating-a-codespace-for-a-**

repository.

- **Google Collaboratory**: Create a Google account, open **https://colab.research.google.com/**, and create a new notebook.

Azure Notebook GitHub Codespace and Google Collaboratory are cloud-based and easy-to-use platforms. To run and set up an environment locally, install the Anaconda distribution on your machine and follow the software installation instructions.

Software installation

Now, let us look at the steps to install Anaconda to run the sample code and tutorials on the local machine as follows:

1. Download the Anaconda Python distribution from the following link: **https://www.anaconda.com/download**
2. Once the download is complete, run the setup to begin the installation process.
3. Once the Anaconda application has been installed, click **Close** and move to the next step to launch the application.

   Check Anaconda installation instructions in the following:
   https://docs.anaconda.com/free/anaconda/install/index.html

Launch application

Now, let us lunch the installed Anaconda navigator and the JupyterLab in it.

Following are the steps:

1. After installing the Anaconda navigator, open any Anaconda navigator and then install and launch JupyterLab.
2. This will start the Jupyter server listening on port 8888.

Usually, a pop-up window comes with a default browser, but you can also start the JupyterLab application on any web browser, Google Chrome preferred, and go to the following URL:

http://localhost:8888/

3. A blank notebook is launched in a new window. You can write Python code on it.
4. Select the cell and press run to execute the code.

The environment is now ready to write, run and execute tutorials.

Basic overview of technology

Python, NumPy, pandas, Sklearn, Matplotlib will be used in most of the tutorials. Let us have a look at them in the following section.

Python

To know more about Python and installation you can refer to the following link: **https://www.python.org/about/gettingstarted/**. Execute Python-version in terminal or command prompt, and if you see the Python version as output, you are good to go, else, install Python. There are different ways to install Python packages on Jupyter Notebook, depending on the package manager you use and the environment you work in, as follows:

- If you use **pip** as your package manager, you can install packages directly from a code cell in your notebook by typing **!pip install <package_name>** and running the cell. Then replace **<package_name>** with the name of the package you want to install.
- If you use **conda** as your package manager, you can

install packages from a JupyterLab cell by typing **!conda install <package_name> --yes** and running the cell. The **--yes** flag is to avoid prompts that asks for confirmation.

- If you want to install a specific version of Python for your notebook, you can use the ipykernel module to create a new kernel with that version. For example, if you have *Python 3.11* and pip installed on your machine, you can type **!pip3.11 install ipykernel** and **!python3.11 -m ipykernel install –user** in two separate code cells and run them. Then, you can select *Python 3.11* as your kernel from the kernel menu.

Further tutorials will be based on the JupyterLab.

pandas

pandas is mainly used for data analysis and manipulation in Python. More can be read at: **https://pandas.pydata.org/docs/**

Following are the ways to install pandas:

- In Jupyter Notebook, execute **pip install pandas**
- In the conda environment, execute **conda install pandas --yes**

NumPy

NumPy is a Python package for numerical computing, multi-dimensional array, and math computation. More can be read at **https://pandas.pydata.org/docs/**.

Following are the ways to install NumPy:

- In Jupyter Notebook, execute **pip install pandas**
- In the conda environment, execute **conda install pandas –yes**

Sklearn

Sklearn is a Python package that provides tools for machine learning, such as data preprocessing, model selection, classification, regression, clustering, and dimensionality reduction. Sklearn is mainly used for predictive data analysis and building machine learning models. More can be read at **https://scikit-learn.org/0.21/documentation.html**.

Following are the ways to install Sklearn:

- In Jupyter Notebook, execute **pip install scikit-learn**
- In the conda environment, execute **conda install scikit-learn –yes**

Matplotlib

Matplotlib is mainly used to create static, animated, and interactive visualizations (plots, figures, and customized visual style and layout) in Python. More can be read at **https://matplotlib.org/stable/index.html**.

Following are the ways to install Matplotlib:

- In Jupyter notebook, excute **pip install matplotlib**
- In the conda environment, execute **conda install matplotlib --yes**

Statistics, data and its importance

As established, statistics is a disciplined approach that enables us to derive insights from diverse forms of data. By applying statistical principles, we can understand what is happening around us quantitatively, evaluate claims, avoid misleading interpretations, produce trustworthy results, and support data-driven decision-making. Statistics also equips us to make predictions and deepen our understanding of the subjects we study.

Data, in turn, serves as the raw material that fuels statistical analysis. It may take various forms—numbers, words,

images, sounds, or videos—and provides the foundational information needed to extract useful knowledge and generate actionable insights. Through careful examination and interpretation, data leads us toward new discoveries, informed decisions, and credible forecasts.

Ultimately, data and statistics are interdependent. Without data, statistics has no basis for drawing conclusions; without statistics, raw data remains untapped and lacks meaning. When combined, they answer fundamental WH questions—Who, What, When, Where, Why, and How—with clarity and confidence, guiding our understanding and shaping the decisions we make.

Types of data

Data can be in different form and type but generally it can be divided into two types, that is, qualitative and quantitative.

Qualitative data

Qualitative data cannot be measured or counted in numbers. Also known as categorical data, it is descriptive, interpretation-based, subjective, and unstructured. It describes the qualities or characteristics of something. It helps to understand the reasoning behind it by asking why, how, or what. It includes nominal and ordinal data. For example, gender of person, race of a person, smartphone brand, hair color type, marital status, and occupation of a person.

Tutorial 1.1: To implement creating a data frame consisting of only qualitative data.

To create a data frame with pandas, **import pandas as pd**, then use the **DataFrame()** function and pass a data source, such as a dictionary, list, or array, as an argument.

```
#Import the pandas library to create a pandas Datafram
e
import pandas as pd
# Sample qualitative data
qualitative_data = {
    'Name': ['John', 'Alice', 'Bob', 'Eve', 'Michael'],
    'City': ['New York', 'Los Angeles', 'Chicago', 'San Fran
cisco', 'Miami'],
    'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
    'Occupation': ['Engineer', 'Artist', 'Teacher', 'Doctor', '
Lawyer'],
    'Race': ['Black', 'White', 'Asian', 'Indian', 'Mongolian'],
    'Smartphone Brand': ['Apple', 'Samsung', 'Xiomi', 'Ap
ple', 'Google']
}
# Create the DataFrame
qualtitative_df = pd.DataFrame(qualitative_data)
# Prints the created DataFrame
print(qualtitative_df)
```

Output:

```
     Name    City      Gender Occupation    Race
Smartphone Brand
0    John    New York    Male    Engineer    Black
Apple
1    Alice  Los Angeles Female  Artist      White
Samsung
2    Bob    Chicago     Male    Teacher     Asian       Xiomi
3    Eve    San FranciscoFemale Doctor      Indian
Apple
4     Michael  Miami     Male    Lawyer
Mongolian    Google
```

Row consisting of numbers 0, 1, 2, 3, and 4 is the index column, not part of the qualitative data. To exclude it from output, hide the index column using **to_string()** as follows:

```
print(qualtitative_df.to_string(index=False))
```

Output:

```
   Name         City Gender Occupation      Race Smartph
one Brand
   John     New York   Male   Engineer     Black        A
pple
  Alice  Los Angeles Female     Artist     White      Sam
sung
    Bob      Chicago   Male    Teacher     Asian      Xio
mi
    Eve San Francisco Female     Doctor    Indian
Apple
Michael        Miami   Male     Lawyer Mongolian
Google
```

While we often think of data in terms of numbers, many other forms such as images, audio, videos, and text they can also represent quantitative information when suitably encoded (e.g., pixel intensity values in images, audio waveforms, or textual features like word counts).

Tutorial 1.2: To implement accessing and creating a data frame consisting of the image data.

In this tutorial, we'll work with the open-source Olivetti faces dataset, which consists of grayscale face images collected at AT&T Laboratories Cambridge between April 1992 and April 1994. Each face is represented by numerical pixel values, making them a form of quantitative data. By organizing this data into a DataFrame, we can easily manipulate, analyze, and visualize it for further insights.

To create a data frame consisting of the *Olivetti faces*

dataset, you can use the following steps:

1. Fetch the **Olivetti faces** dataset from sklearn using the **sklearn.datasets.fetch_olivetti_faces** function. This will return an object that holds the data and some metadata.
2. Use the **pandas.DataFrame** constructor to create a data frame from the data and the feature names. You can also add a column for the target labels using the target and **target_names** attributes of the object.
3. Use the pandas method to display and analyze the data frame. For example, you can use **df.head()**, **df.describe()**, **df.info()**.

```
import pandas as pd
#Import datasets from the sklearn library
from sklearn import datasets
# Fetch the Olivetti faces dataset
faces = datasets.fetch_olivetti_faces()
# Create a dataframe from the data and feature nam
es
df = pd.DataFrame(faces.data)
# Add a column for the target labels
df["target"] = faces.target
# Display the first 3 rows of the dataframe
print(f"{df.head(3)}")
# Print new line
print("\n")
# Display the first image in the dataset
import matplotlib.pyplot as plt
plt.imshow(df.iloc[0, :-1].values.reshape(64, 64), cm
ap="gray")
plt.title(f"Image of person {df.iloc[0, -1]}")
```

```
plt.show()
```

Quantitative data

Quantitative data is measurable and can be expressed numerically. It is useful for statistical analysis and mathematical calculations. For example, if you inquire about the number of books people have read in a month, their responses constitute quantitative data. They may reveal that they have read, let us say, **three books**, **zero books**, or **ten books**, providing information about their reading habits. Quantitative data is easily comparable and allows for calculations. It can provide answers to questions such as **How many? How much? How often?** and **How fast?**

Tutorial 1.3: To implement creating a data frame consisting of only quantitative data is as follows:

```
#Import the pandas library to create a pandas Datafram
e
import pandas as pd
quantitative_df = pd.DataFrame({
    "price": [300000, 250000, 400000, 350000, 450000],
    "distance": [10, 15, 20, 25, 30],
    "height": [170, 180, 190, 160, 175],
    "weight": [70, 80, 90, 60, 75],
    "salary": [5000, 6000, 7000, 8000, 9000],
    "temperature": [25, 30, 35, 40, 45],
})
# Print the DataFrame without index
print(quantitative_df.to_string(index=False))
```

Output:

```
 price  distance  height  weight  salary  temperature
300000        10     170      70    5000           25
```

3.	250000	15	180	80	6000	30
4.	400000	20	190	90	7000	35
5.	350000	25	160	60	8000	40
6.	450000	30	175	75	9000	45

Tutorial 1.4: To implement accessing and creating a data frame by loading the tabular **iris** data.

Iris tabular dataset contains 150 samples of iris flowers with four features, that is, sepal length, sepal width, petal length, and petal width and three classes, that is, setosa, versicolor, and virginica. The sepal length, sepal width, petal length, petal width, and target (class) are columns of the table[1].

To create a data frame consisting of the **iris** dataset, you can use the following steps:

1. First, you need to load the **iris** dataset from sklearn using the **sklearn.datasets.load_iris** function. This will return a bunch object that holds the data and some metadata.
2. Next, you can use the **pandas.DataFrame** constructor to create a data frame from the data and the feature names. You can also add a column for the target labels using the target and **target_names** attributes of the bunch object.
1. Finally, you can use the panda method to display and analyze the data frame. For example, you can use **df.head()**, **df.describe()**, **df.info()** as follows:

```
1. import pandas as pd
2. # Import dataset from sklean
3. from sklearn import datasets
4. # Load the iris dataset
5. iris = datasets.load_iris()
6. # Create a dataframe from the data and feature nam
```

```
es
df = pd.DataFrame(iris.data, columns=iris.feature_names)
# Add a column for the target labels
df["target"] = iris.target
# Display the first 5 rows of the dataframe
df.head()
```

Level of measurement

Level of measurement is a way of classifying data based on how precise it is and what we can do with it. Generally, they are four, that is, nominal, ordinal, interval and ratio. Nominal is a category with no inherent order, such as colors. Ordinal is a category with a meaningful order, such as education levels. Interval is equal intervals but no true zero, such as temperature in degrees Celsius, and ratio are equal intervals with a true zero, such as age in years.

Nominal data

Nominal data is qualitative data that does not have a natural ordering or ranking. For example, gender, religion, ethnicity, color, brand ownership of electronic appliances, and person's favorite meal.

Tutorial 1.5: To implement creating a data frame consisting of qualitative nominal data, is as follows:

```
#Import the pandas library to create a pandas Datafram
e
import pandas as pd
nominal_data = {
    "Gender": ["Male", "Female", "Male", "Female", "Male
"],
    "Religion": ["Hindu", "Muslim", "Christian", "Buddhist
```

```
", "Jewish"],
    "Ethnicity": ["Indian", "Pakistani", "American", "Chinese", "Israeli"],
    "Color": ["Red", "Green", "Blue", "Yellow", "White"],
    "Electronic Appliances Ownership": ["Samsung", "LG", "Apple", "Huawei", "Sony"],
    "Person Favorite Meal": ["Biryani", "Kebab", "Pizza", "Noodles", "Falafel"],
    "Pet Preference": ["Dog", "Cat", "Parrot", "Fish", "Hamster"]
}
# Create the DataFrame
nominal_df = pd.DataFrame(nominal_data)
# Display the DataFrame
print(nominal_df)
```

Output:

```
  Gender  Religion  Ethnicity  Color Electronic Appliances Ownership  \
0   Male     Hindu     Indian    Red                          Samsung
1 Female    Muslim   Pakistani  Grezn                               LG
2   Male  Christian   American   Blue                            Apple
3 Female  Buddhist    Chinese  Yellow                           Huawei
4   Male    Jewish    Israeli  Whie                             Sony

  Person Favorite Meal Pet Preference
0              Biryani            Dog
1                Kebab            Cat
```

```
2    Pizza    Parrot
3  Noodles      Fish
4  Falafel   Hamster
```

Ordinal data

Ordinal data is qualitative data that has a natural ordering or ranking. For example, student ranking in class (1st, 2nd, or 3rd), educational qualification (high school, undergraduate, or graduate), satisfaction level (bad, average, or good), income level range, level of agreement (agree, neutral, or disagree).

Tutorial 1.6: To implement creating a data frame consisting of qualitative ordinal data is as follows:

```
import pandas as pd
ordinal_data = {
    "Student Rank in a Class": ["1st", "2nd", "3rd", "4th", "5th"],
    "Educational Qualification": ["Graduate", "Undergraduate", "High School", "Graduate", "Undergraduate"],
    "Satisfaction Level": ["Good", "Average", "Bad", "Average", "Good"],
    "Income Level Range": ["80,000-100,000", "60,000-80,000", "40,000-60,000", "100,000-120,000", "50,000-70,000"],
    "Level of Agreement": ["Agree", "Neutral", "Disagree", "Neutral", "Agree"]
}
ordinal_df = pd.DataFrame(ordinal_data)
print(ordinal_df)
```

Output:

```
  Student Rank in a Class Educational Qualification Satisfaction Level  \
0                     1st                  Graduate               Good
```

3.	1	2nd	Undergraduate	Average
4.	2	3rd	High School	Bad
5.	3	4th	Graduate	Average
6.	4	5th	Undergraduate	Good

8.		Income Level Range	Level of Agreement
9.	0	80,000-100,000	Agree
10.	1	60,000-80,000	Neutral
11.	2	40,000-60,000	Disagree
12.	3	100,000-120,000	Neutral
13.	4	50,000-70,000	Agree

Discrete data

Discrete data is quantitative data, integers or whole numbers, they cannot be subdivided into parts. For example, total number of students present in a class, cost of a cell phone, number of employees in a company, total number of players who participated in a competition, days in a week, number of books in a library, etc. For example, number of coins in a jar, it can only be a whole number like 1,2,3 and so on.

Tutorial 1.7: To implement creating a data frame consisting of quantitative discrete data is as follows:

```
1. import pandas as pd
2. discrete_data = {
3.     "Students": [25, 30, 35, 40, 45],
4.     "Cost": [500, 600, 700, 800, 900],
5.     "Employees": [100, 150, 200, 250, 300],
6.     "Players": [50, 40, 30, 20, 10],
7.     "Week": [7, 7, 7, 7, 7]
8. }
```

```
discrete_df = pd.DataFrame(discrete_data)
discrete_df
```

Output:

```
 Students Cost Employees Players Week
0 25 500 100 50 7
1 30 600 150 40 7
2 35 700 200 30 7
3 40 800 250 20 7
4 45 900 300 10 7
```

Continuous data

Continuous data is quantitative data that can take any value (including fractional value) within a range and have no gaps between them. No gaps mean that if a person's height is 1.75 meters, there is always a possibility of height being between 1.75 and 1.76 meters, such as 1.751 or 1.755 meters.

Interval data

Interval data is quantitative numerical data with inherent order. They always have an arbitrary zero, an arbitrary zero meaning no meaningful zero, chosen by convention, not by nature. For example, a temperature of zero degrees Fahrenheit does not mean that there is no heat or temperature, here, zero is an arbitrary zero point. For example, temperature (Celsius or Fahrenheit), GMAT score (200-800), SAT score (400-1600).

Tutorial 1.8: To implement creating a data frame consisting of quantitative interval data is as follows:

```
import pandas as pd
interval_data = {
    "Temperature": [10, 15, 20, 25, 30],
    "GMAT_Score": [600, 650, 700, 750, 800],
```

```
    "SAT_Score (400 - 1600)": [1200, 1300, 1400, 1500, 1600],
    "Time": ["9:00", "10:00", "11:00", "12:00", "13:00"]
}
interval_df = pd.DataFrame(interval_data)
# Print DataFrame as it is without print() also
interval_df
```

Output:

```
 Temperature GMAT_Score SAT_Score (400 - 1600) Time
0 10 600 1200 9:00
1 15 650 1300 10:00
2 20 700 1400 11:00
3 25 750 1500 12:00
4 30 800 1600 13:00
```

Ratio data

Ratio data is naturally, numerical ordered data with an absolute, where zero is not arbitrary but meaningful. For example, height, weight, age, tax amount has true zero point that is fixed by nature, and they are measured on a ratio scale. Zero height means no height at all, like a point in space. There is nothing shorter than zero height. Zero tax amount means no tax at all, like being exempt. There is nothing lower than zero tax amount.

Tutorial 1.9: To implement creating a data frame consisting of quantitative ratio data is as follows:

```
import pandas as pd
ratio_data = {
    "Height": [170, 180, 190, 200, 210],
    "Weight": [60, 70, 80, 90, 100],
    "Age": [20, 25, 30, 35, 40],
```

```
    "Speed": [80, 90, 100, 110, 120],
    "Tax Amount": [1000, 1500, 2000, 2500, 3000]
}
ratio_df = pd.DataFrame(ratio_data)
ratio_df
```

Output:

```
 Height Weight Age Speed Tax Amount
0 170 60 20 80 1000
1 180 70 25 90 1500
2 190 80 30 100 2000
3 200 90 35 110 2500
4 210 100 40 120 3000
```

Tutorial 1.10: To implement loading the ratio data in a JSON format and displaying it.

Sometimes, data can be in **JSON**. The data used in the following Tutorial 1.10 is in **JSON** format. In that case **json.loads()** method can load it. **JSON** is a text format for data interchange based on JavaScript as follows:

```
# Import json
import json
# The JSON string:
json_data = """
[
  {
    "Height": 170,
    "Weight": 60,
    "Age": 20,
    "Speed": 80,
    "Tax Amount": 1000
  },
  {
```

```
    "Height": 180,
    "Weight": 70,
    "Age": 25,
    "Speed": 90,
    "Tax Amount": 1500
  },
  {
    "Height": 190,
    "Weight": 80,
    "Age": 30,
    "Speed": 100,
    "Tax Amount": 2000
  },
  {
    "Height": 200,
    "Weight": 90,
    "Age": 35,
    "Speed": 110,
    "Tax Amount": 2500
  },
  {
    "Height": 210,
    "Weight": 100,
    "Age": 40,
    "Speed": 120,
    "Tax Amount": 3000
   }
]
"""
# Convert to Python object (list of dicts):
```

```
data = json.loads(json_data)
data
```

Output:

```
[{'Height': 170, 'Weight': 60, 'Age': 20, 'Speed': 80, 'Tax Amount': 1000},
 {'Height': 180, 'Weight': 70, 'Age': 25, 'Speed': 90, 'Tax Amount': 1500},
 {'Height': 190, 'Weight': 80, 'Age': 30, 'Speed': 100, 'Tax Amount': 2000},
 {'Height': 200, 'Weight': 90, 'Age': 35, 'Speed': 110, 'Tax Amount': 2500},
 {'Height': 210, 'Weight': 100, 'Age': 40, 'Speed': 120, 'Tax Amount': 3000}]
```

Distinguishing qualitative and quantitative data

As discussed above, qualitative data describes the quality or nature of something, such as color, shape, taste, or opinion etc. whereas quantitative data is the data that measures the quantity or amount of something, such as length, weight, speed, or frequency. Qualitative data can be further classified as nominal (categorical) or ordinal (ranked). Quantitative data can be further classified as discrete (countable) or continuous (measurable).The following methods are used to understand if data is qualitative or quantitative in nature.

dtype(): It is used to check the data types of the data frame.

Tutorial 1.11: To implement **dtype()** to check the datatypes of the different features or column in a data frame, as follows:

```
import pandas as pd
# Create a data frame with qualitative and quantitative
```

```
columns
df = pd.DataFrame({
    "age": [25, 30, 35], # a quantitative column
    "gender": ["female", "male", "male"], # a qualitative column
    "hair color": ["black", "brown", "white"], # a qualitative column
    "marital status": ["single", "married", "divorced"], # a qualitative column
    "salary": [5000, 6000, 7000], # a quantitative column
    "height": [6, 5.7, 5.5], # a quantitative column
    "weight": [60, 57, 55] # a quantitative column
})
# Print the data frame
print(df)
# Print the data types of each column using dtype()
print(df.dtypes)
```

Output:

```
   age  gender hair color marital status  salary  height  weight
0   25  female      black         single    5000     6.0      60
1   30    male      brown        married    6000     5.7      57
2   35    male      white       divorced    7000     5.5      55
age                int64
gender            object
hair color        object
marital status    object
salary             int64
height           float64
weight             int64
```

describe(): You can also use the describe method from pandas to generate descriptive statistics for each column. This method will only show statistics for quantitative columns by default, such as mean, standard deviation, minimum, maximum, etc.

You need to specify **include='O'** as an argument to include qualitative columns. This will show statistics for qualitative columns, such as count, unique values, top values, and frequency. As you can see, the descriptive statistics for qualitative and quantitative columns are different, reflecting the nature of the data.

Tutorial 1.12: To implement **describe()** in the data frame used in *Tutorial 1.11* of **dtype(),** is as follows:

```
# Print the descriptive statistics for quantitative columns
print(df.describe())
# Print the descriptive statistics for qualitative columns
print(df.describe(include='O'))
```

Output:

```
       age  salary    height     weight
count   3.0     3.0  3.000000   3.000000
mean   30.0  6000.0  5.733333  57.333333
std     5.0  1000.0  0.251661   2.516611
min    25.0  5000.0  5.500000  55.000000
25%    27.5  5500.0  5.600000  56.000000
50%    30.0  6000.0  5.700000  57.000000
75%    32.5  6500.0  5.850000  58.500000
max    35.0  7000.0  6.000000  60.000000
       gender hair color marital status
count       3          3              3
unique      2          3              3
top      male      black         single
freq        2          1              1
```

value_counts(): To count unique values in a data frame, the **value_counts()** is used. It also displays the data type **dtype**. The **dtype** displays is the data type of the values in the series object returned by the **value_counts** method.

Tutorial 1.13: To implement **value_count()** to count unique value in a data frame as follows:

```
# To count the values in `gender` column
print(df['gender'].value_counts())
print("\n")
# To count the values in `age` column
print(df['age'].value_counts())
```

Output:

```
gender
male      2
female    1
Name: count, dtype: int64

age
25    1
30    1
35    1
```

In above *Tutorial 1.13* of **value_counts(),** the values are counts of each unique value in the gender column of the data frame, and the data type is **int64**, which means 64-bit integer.

is_numeric_dtype(), is_string_dtype(): These functions from the **pandas.api.types** module can help you determine if a column contains numeric or string (object) data.

Tutorial 1.14: To implement checking the numeric and string data type of a data frame column with **is_numeric_dtype()** and **is_string_dtype()** functions is as follows:

```
# Import module for data type checking and inference.
import pandas.api.types as ptypes
# Checks if the column 'hair color' in df is of the string d type and prints the result
print(f"Is string?:  {ptypes.is_string_dtype(df['hair color' ])}")
# Checks if the column 'weight' in df is of the numeric dt ype and prints the result
print(f"Is numeric?:  {ptypes.is_numeric_dtype(df['weig ht'])}")
# Checks if the column 'salary' in df is of the string dtyp e and prints the result
print(f"Is string?:  {ptypes.is_string_dtype(df['salary'])}" )
```

Output:

```
Is string?:  True
Is numeric?:  True
Is string?:  False
```

Also, in *Tutorial 1.14* we can use a for loop to check it for all the columns iteratively as follows:

```
# Check the data types of each column using is_numeric _dtype() and is_string_dtype()
for col in df.columns:
    print(f"{col}:")
    print(f"Is numeric? {ptypes.is_numeric_dtype(df[col]) }")
    print(f"Is string? {ptypes.is_string_dtype(df[col])}")
    print()
```

info(): It describes the data frame with a column name, the number of not null values, and the data type of each column.

Tutorial 1.15: To implement **info()** to view the information

about a data frame is as follows:

```
df.info()
```

The output will display the summary consisting of column names, **non-null** count values, data types of each column, and many more.

```
RangeIndex: 3 entries, 0 to 2
Data columns (total 7 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             3 non-null      int64
 1   gender          3 non-null      object
 2   hair color      3 non-null      object
 3   marital status  3 non-null      object
 4   salary          3 non-null      int64
 5   height          3 non-null      float64
 6   weight          3 non-null      int64
dtypes: float64(1), int64(3), object(3)
memory usage: 296.0+ bytes
```

head() and tail(): head() displays the data frame from the top, and the **tail()** displays it from the last or bottom.

Tutorial 1.16: To implement **head()** and **tail()** to view the top and bottom rows of data frame respectively.

head() displays the data frame with the first few rows. Inside the parenthesis of **head()**, we can define the number of rows we want to view. For example, to view the first ten rows of the data frame, write **head(10)**. Same with **tail()** also as follows:

```
# View the first few rows of the DataFrame
print(df.head())
# View first 2 rows of the DataFrame
df.head(1)
```

The output will display the data frame from the top, and **head(1)** will display the topmost row of data frame as follows:

```
  age  gender hair color marital status  salary  height  weight
0   25  female      black        single    5000     6.0      60
1   30    male      brown       married    6000     5.7      57
2   35    male      white      divorced    7000     5.5      55
age gender hair color marital status salary height weigh
t
0 25 female black single 5000 6.0 60
```

Tutorial 1.17: To implement **tail()** and display the bottom most rows of data frame is as follows:

```
# Import display function from IPython module to displa
y the dataframe
from IPython.display import display
# View the last few rows of the DataFrame
display(df.tail())
# View last 2 rows of the DataFrame
display(df.tail(1))
```

In output **tail(1)** will display the bottommost row of data frame as follows:

```
 age gender hair color marital status salary height weig
ht
0 25 female black single 5000 6.0 60
1 30 male brown married 6000 5.7 57
2 35 male white divorced 7000 5.5 55
age gender hair color marital status salary height weigh
t
2 35 male white divorced 7000 5.5 55
```

Other methods: Besides function described above there

are few other methods in Python that are useful to distinguish qualitative and quantitative data. They are as follows:

- **select_dtypes(include='____'):** It is used to select columns with specific datatype that is, number and object.

```
# Select and display DataFrame with only numeric values
display(df.select_dtypes(include='number'))
```

The output will display only the numeric column of the data frame as follows:

```
 age salary height weight
0 25 5000 6.0 60
1 30 6000 5.7 57
2 35 7000 5.5 55
```

To select only object data types, an object is used. In pandas, objects are the column containing strings or mixed types of data. It is the default data type for columns that have text or arbitrary Python objects as follows:

```
# Select and display DataFrame with only object values in the same cell(display() displays DataFrame in same cell)
display(df.select_dtypes(include='object'))
```

Output will include only object type columns as follows:

```
 gender hair color marital status
0 female black single
1 male brown married
2 male white divorced
```

- **groupby(): groupby()** is used to group based on column as follows:

```
# Group by gender
```

```
2. df.groupby("gender")
```

After grouping based on the column name, it can be used to display the summary statistics of a grouped data frame. **describe()** method on the **groupby()** object can be used to find summary statistics of each group as follows:

```
1. # Describe dataframe summary statistics by gender
2. df.groupby('gender').describe()
```

Further to print the count of each group, you can use the **size()** or **count()** method on the **groupby()** object as follows:

```
1. # Print count of group object with size
2. print(df.groupby('gender').size())
3. # Print count of group object with count
4. print(df.groupby('gender').count())
```

Output:

```
1. gender
2. female    1
3. male      2
4. dtype: int64
5.   age  hair color  marital status  salary  height  weigh
   t
6. gender
7. female    1   1                1       1       1       1
8. male      2   2                2       2       2       2
```

- **groupby().sum(): groupby().sum()** groups data and then display sum in each group as follows:

```
1. # Group by gender and hair color and calculate the
   sum of each group
2. df.groupby(["gender", "hair color"]).sum()
```

- **columns**: **columns** display column names. Sometimes, through descriptive columns names, types of data can

be distinguished. So, displaying column name can be useful as follows:

```
# Displays all column names.
df.columns
```

- **type(): type()** is used to display the type of a variable. It can be used to determine the type of a single variable as follows:

```
# Declare variable
x = 42
y = "Hello"
z = [1, 2, 3]
# Print data types
print(type(x))
print(type(y))
print(type(z))
```

Tutorial 1.18: To implement **read_json(),** to read and view nobel prize dataset in JSON format.

Let us load a *nobel prizedataset*[2] and see what kind of data it contains. The *Tutorial 1.18* flattens nested **JSON** data structures into a data frame as follows:

```
import pandas as pd
# Read the json file from the direcotory
json_df = pd.read_json("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.json")
# Convert the json data into a dataframe
data = json_df["prizes"]
prize_df = pd.json_normalize(data)
# Display the dataframe
prize_df
```

To see what type of data **prize_df** contains use **info()** and **head()**, is as follows:

```
prize_df.info()
```

```
2. prize_df.head()
```

Alternatively, to *Tutorial 1.18*, the *nobel prize dataset*[3] can be accessed directly by sending the request as shown in the following code:

```
1. import pandas as pd
2. # Send HTTP requests using Python
3. import requests
4. # Get the json data from the url
5. response = requests.get("https://api.nobelprize.org/v1/prize.json")
6. data = response.json()
7. # Convert the json data into a dataframe
8. prize_json_df = pd.json_normalize(data, record_path="prizes")
9. prize_json_df
```

Tutorial 1.19: To implement **read_csv(),** to read and view nobel prize dataset in CSV format is as follows:

```
1. import pandas as pd
2. # Read the json file from the direcotory
3. prize_csv_df = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.csv")
4. # Display the dataframe
5. prize_csv_df
```

Tutorial 1.20: To implement use of NumPy and to read diabetes dataset in CSV files.

Most common ways are using **numpy.loadtxt()** and using **numpy.genfromtxt()**. **numpy.loadtxt()** assumes that the file has no missing values, no comments, and no headers and uses whitespace as the delimiter by default. We can change the delimiter to a comma by passing **delimiter =**',' as a parameter. Here, the CSV file has one header row, which is a string, so we use **skiprows = 1** this skips the

first row of the CSV file and loads the rest of the data as a NumPy array as follows:

```
1. import numpy as np
2. arr = np.loadtxt('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv', delimiter=',', skiprows=1)
3. print(arr)
```

The **numpy.genfromtxt()** function can handle missing values, comments, headers, and various delimiters. We can use the **missing_values** parameter to specify which values to treat as missing. We can use the comments parameter to specify which character indicates a comment line, such as # or %. For example, if you have a CSV file named **diabetes.csv** that looks as follows:

```
1. import numpy as np
2. arr = np.genfromtxt('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv', delimiter=',', names=True, missing_values='?', dtype=None)
3. print(arr)
```

Univariate, bivariate, and multivariate data

Univariate, bivariate, and multivariate data are terms used in statistics to describe the number of variables and their relationships within a dataset. Where univariate means one, bivariate means two and multivariate is more than two. These concepts are fundamental to statistical analysis and play a crucial role in various fields, from social sciences to natural sciences, engineering and beyond.

Univariate data and univariate analysis

Univariate analysis involves observing only one variable or attribute. For example, height of students in a class, color of cars in a parking lot, or salary of employees in a

company are all univariate data. Univariate analysis analyzes only one variable column or attribute at a time. For example, analyzing only the patient height column at a time or the person's salary column.

Tutorial 1.21: To implement univariate data and univariate analysis by selecting a column or variable or attribute from the CSV dataset and compute its mean, standard deviation, frequency or distribution with other information using **describe()** as follows:

```
1. import pandas as pd
2. from IPython.display import display
3. diabities_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')
4. #To view all the column names
5. print(diabities_df.columns)
6. # Select the column Glucose column as a DataFrame from diabities_df DataFrame
7. display(diabities_df[['Glucose']])
8. # describe() gives the mean,standard deviation
9. print(diabities_df[['Glucose']].describe())
```

In the above, we selected only the **Glucose** column of the **diabities_df** and analyzed only that column. This kind of single-column analysis is univariate analysis.

Tutorial 1.22: To further implement computation of median, mode range, frequency or distribution of variables in continuation with *Tutorial 1.21,* is as follows:

```
1. # Use mode() for computing most frequest value i.e, mode
2. print(diabities_df[['Glucose']].mode())
3. # To get range simply subtract DataFrame maximum value by the DataFrame minimum value. Use df.max() and df.min() for maximum and minimum value
4. mode_range = diabities_df[['Glucose']].max() - diabities
```

```
_df[['Glucose']].min()
print(mode_range)
# For frequency or distribution of variables use value_counts()
diabities_df[['Glucose']].value_counts()
```

Bivariate data

Bivariate data consists of observing two variables or attributes for each individual or unit. For example, if you wanted to study the relationship between the age and height of students in a class, you would collect the age and height of each student. Age and height are two variables or attributes, and each student is an individual or unit. Bivariate analysis analyzes how two different variables, columns, or attributes are related. For example, the correlation between people's height and weight or between hours worked and monthly salary.

Tutorial 1.23: To implement bivariate data and bivariate analysis by selecting two columns or variables or attributes from the CSV dataset and to describe them, as follows:

```
import pandas as pd
from IPython.display import display
diabities_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')
#To view all the column names
print(diabities_df.columns)
# Select two column Glucose column as a DataFrame from diabities_df DataFrame
display(diabities_df[['Glucose','Age']])
# describe() gives the mean,standard deviation
print(diabities_df[['Glucose','Age']].describe())
# Use mode() for computing most frequest value i.e, mo
```

```
de
print(diabities_df[['Glucose']].mode())
# To get range simply subtract DataFrame maximum value by the DataFrame minimum value. Use df.max() and df.min() for maximum and minimum value
mode_range = diabities_df[['Glucose']].max() - diabities_df[['Glucose']].min()
print(mode_range)
# For frequency or distribution of variables use value_counts()
diabities_df[['Glucose']].value_counts()
```

Here, we compared two columns, **glucose** and **age** in **diabities_df** data frame, which involved multiple data frame columns making it bivariate analysis.

Alternatively two or more columns can be accessed using **loc[row_start:row:stop,column_start:column:stop]** or also through column index via slicing by using **iloc[row_start:row:stop,column_start:column:stop]** as follows:

```
# Using loc
diabities_df.loc[:, ['Glucose','Age']]
# Using iloc, column index and slicing
diabities_df.iloc[:,0:2]
```

Further, to compute the correlation between two variables or two columns, such as **glucose** and **age**, we can use columns along with **corr()** as follows:

```
diabities_df['Glucose'].corr(diabities_df['Age'])
```

Correlation is a statistical measure that indicates how two variables are related to each other. A positive correlation means that the variables increase or decrease together, while a negative correlation means that the variables move in opposite directions. A correlation value close to zero

means that there is no linear relationship between the variables.

In the context of **diabetes_df['Glucose'].corr(diabetes_df['Age'])**, the random positive correlation value of **0.26** means that there is a weak positive correlation between glucose level and age in the diabetes dataset. This implies that older people tend to have higher glucose levels than younger people but the relationship is not very strong or consistent. Correlation can be computed using different methods such as **pearson**, **kendall**, or **spearman** then specify method **='__'** in **corr()** as follows:

```
1. diabities_df['Glucose'].corr(diabities_df['Age'], method=
   'kendall')
```

Multivariate data

Multivariate data consists of observing three or more variables or attributes for each individual or unit. For example, if you want to study the relationship between the age, gender, and income of customers in a store, you would collect this data for each customer. Age, gender, and income are the three variables or attributes, and each customer is an individual or unit. In this case, the data you collect will be multivariate data because it requires observations on three variables or attributes for each individual or unit. For example, the correlation between age, gender, and sales in a store or between temperature, humidity, and air quality in a city.

Tutorial 1.24: To implement multivariate data and multivariate analysis by selecting multiple columns or variables or attributes from the CSV dataset and describe them, as follows:

```
1. import pandas as pd
2. from IPython.display import display
```

```
3. diabities_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')
4. #To view all the column names
5. print(diabities_df.columns)
6. # Select the column Glucose column as a DataFrame from diabities_df DataFrame
7. display(diabities_df[['Glucose','BMI', 'Age', 'Outcome']])
8. # describe() gives the mean,standard deviation
9. print(diabities_df[['Glucose','BMI', 'Age', 'Outcome']].describe())
```

Alternatively, multivariate analysis can be performed by describing the whole data frame as follows:

```
1. # describe() gives the mean,standard deviation
2. print(diabities_df.describe())
3. # Use mode() for computing most frequest value i.e, mode
4. print(diabities_df.mode())
5. # To get range simply subtract DataFrame maximum value by the DataFrame minimum value. Use df.max() and df.min() for maximum and minimum value
6. mode_range = diabities_df.max() - diabities_df.min()
7. print(mode_range)
8. # For frequency or distribution of variables use value_counts()
9. diabities_df.value_counts()
```

Further, to compute the correlation between all the variables in the data frame, use **corr()** after the data frame variable name as follows:

```
1. diabities_df.corr()
```

You can also apply various multivariate analysis techniques, as follows:

- **Principal Component Analysis (PCA)**: It transforms

high-dimensional data into a smaller set of uncorrelated variables (principal components) that capture the most variance, thereby simplifying the dataset while retaining essential information. It makes easier to visualize, interpret, and model multivariate relationships

    - **Library**: Scikit-learn
    - **Method**: **PCA(n_components=___)**
- **Multivariate regression**: This is used to analyze the relationship between multiple dependent and independent variables.
    - **Library**: Statsmodels
    - **Method**: **statsmodels.api.OLS** for ordinary least squares regression. It allows you to perform multivariate linear regression and analyze the relationship between multiple dependent and independent variables. Regression can also be performed using scikit-learn's **LinearRegression()**, **LogisticRegression()**, and many more.
- **Cluster analysis**: This is used to group similar data points together based on their characteristics.
    - **Library**: Scikit-learn
    - **Method**: **sklearn.cluster. KMeans** for K-means clustering. It allows you to group similar data points together based on their characteristics. And many more.
- **Factor analysis**: This is used to identify underlying latent variables that explain the observed variance.
    - **Library**: FactorAnalyzer
    - **Method**: **FactorAnalyzer** for factor analysis. It allows you to perform **Exploratory Factor**

Analysis (EFA) to identify underlying latent variables that explain the observed variance.

- **Canonical Correlation Analysis (CCA):** To explore the relationship between two sets of variables.
    - **Library**: Scikit-learn
    - **Method**: **sklearn.cross_decomposition** and CCA allows you to explore the relationship between two sets of variables and find linear combinations that maximize the correlation between the two sets.

Tutorial 1.25: To implement **Principal Component Analysis (PCA)** for dimensionality reduction is as follows:

```
import pandas as pd
# Import principal component analysys
from sklearn.decomposition import PCA
# Scales data between 0 and 1
from sklearn.preprocessing import StandardScaler
# Import matplotlib to plot visualization
import matplotlib.pyplot as plt
# Step 1: Load your dataset into a DataFrame
# Assuming you have your dataset stored in a CSV file called "data.csv", load it into a Pandas DataFrame.
data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
# Step 2: Separate the features and the outcome variable (if applicable)
# If the "Outcome" column represents the dependent variable and not a feature, you should separate it from the features.
# If it's not the case, you can skip this step.
X = data.drop("Outcome", axis=1)  # Features
```

```
y = data["Outcome"]  # Outcome (if applicable)
# Step 3: Standardize the features
# PCA is sensitive to the scale of features, so it's crucial to standardize them to have zero mean and unit variance.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
# Step 4: Apply PCA for dimensionality reduction
# Create a PCA instance and specify the number of components you want to retain.
# If you want to reduce the dataset to a certain number of dimensions (e.g., 2 or 3), set the 'n_components' accordingly.
pca = PCA(n_components=2)  # Reduce to 2 principal components
X_pca = pca.fit_transform(X_scaled)
# Step 5: Explained Variance Ratio
# The explained variance ratio gives us an idea of how much information each principal component captures.
explained_variance_ratio = pca.explained_variance_ratio_
# Step 6: Visualize the Explained Variance Ratio
plt.bar(range(len(explained_variance_ratio)), explained_variance_ratio)
plt.xlabel("Principal Component")
plt.ylabel("Explained Variance Ratio")
plt.title("Explained Variance Ratio for Each Principal Component")
# Show the figure
plt.savefig('skew_negative.jpg',dpi=600,bbox_inches='tight')
```

```
35. plt.show()
```

PCA reduces the dimensions but it also results in some loss of information as we only retain the most important components. Here, the original 8-dimensional diabetes data set has been transformed into a new 2-dimensional data set. The two new columns represent the first and second principal components, which are linear combinations of the original features. These principal components capture the most significant variation in the data.

The columns of the data set pregnancies, glucose, blood pressure, skin thickness, insulin, BMI, diabetes pedigree function, and age are reduced to 2 principal components because we specify **n_components=2** as shown in *Figure 1.1*.

Output:

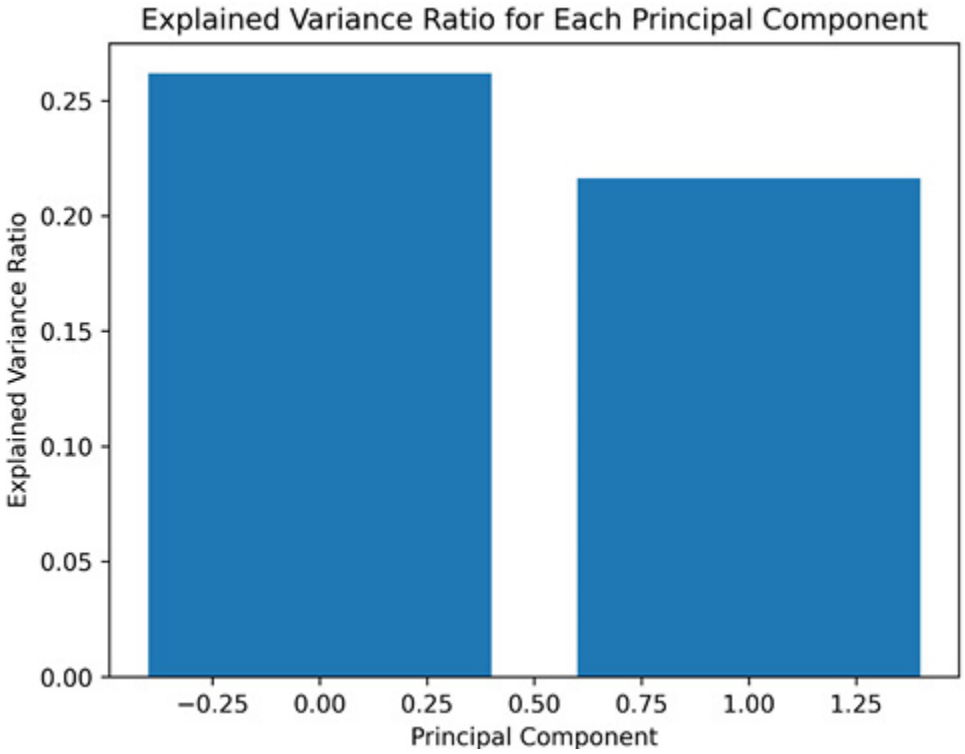

Figure 1.1: *Explained variance ratio for each principal component*

Following is what you can infer from these explained

variance ratios in this **diabetes dataset**:

- **The First Principal Component (PC1):** With an explained variance of 0.27, PC1 captures the largest portion of the data's variability. It represents the direction in the data space along which the data points exhibit the most significant variation. PC1 is the principal component that explains the most significant patterns in the data.
- **The Second Principal Component (PC2):** With an explained variance of 0.23, PC2 captures the second-largest portion of the data's variability. PC2 is orthogonal (uncorrelated) to PC1, meaning it represents a different direction in the data space from PC1. PC2 captures additional patterns that are not explained by PC1 and provides complementary information. PC1 and PC2 account for approximately 50% (0.27 + 0.23) of the total variance.

  You can do similar with NumPy and JSON. Also, you can create different types of plots and charts for data analysis using Matplotlib and Seaborn libraries

Data sources, methods, populations, and samples

Data sources provide information for analysis from surveys, databases, or experiments. Collection methods determine how data is gathered, through interviews, questionnaires, or observations. Population is the entire group being studied, while samples are representative subsets used to draw conclusions with less analysis.

Data source

Data can be primary and secondary. It can be of two types, that is, statistical sources like surveys, census, experiments, and statistical reports and non-statistical

sources like business transactions, social media posts, weblogs, data from wearables and sensors, or personal records.

Tutorial 1.26: To implement reading data from different sources and view statistical and non-statistical data is as follows:

```
import pandas as pd
# To import urllib library for opening and reading URLs
import urllib.request
# To access CSV file replace file name
df = pd.read_csv('url_to_csv_file.csv')
```

To access or read data from different sources, pandas provides **read_csv()** and **read_json()** and **loadtxt()**, **genfromtxt()** in NumPy and many others. The URL can also be used like **https://api.nobelprize.org/v1/prize.json**, but it should be accessible. Most data server would need authentication to access the server.

To read JSON files replace file name in the script as follows:

```
# To access JSON data replace file name
df = pd.read_json('your_file_name.json')
```

To read XML file from a server with NumPy, you can use the **np.loadtxt()** function and pass as an argument a file object created using the **urllib.request.urlopen()** function from the **urllib.request** module. You must also specify the delimiter parameter as **<** or **>** to separate XML tags from the data values. To read XML file, replace files names with appropriate one in the script as follows:

```
# To access and read the XML file using URL
file = urllib.request.urlopen('your_url_to_accessible_xml_file.xml')
# To open the XML file from the URL and store it in a file
```

```
object
```

```
arr = np.loadtxt(file, delimiter='<')
print(arr)
```

Collection methods

Collection methods are surveys, interviews, observations, focus groups, experiments, and secondary data analysis. It can be quantitative, based on numerical data and statistical analysis, or qualitative, based on words, images, actions, and interpretive analysis. Also, sometimes mixed methods, which combine qualitative and quantitative, can be used.

Population and sample

Population is entire group of people, items, or elements you want to study or draw conclusions about. For example, if you want to know the average score of all students in a school, the population is all students. The sample is a subset of the population from which you select and collect data. For example, 20 randomly chosen students from this school are a population sample.

Let us see an example of selecting population and sample using random modules. The **random** module **rand()** function can be utilized to randomly choose unstructured and semi-structured datasets or files. This approach ensures that each data point has an equal probability of being included in the sample, thereby minimizing selection bias and ensuring the sample's representativeness of the broader population.

Tutorial 1.27: To implement **rand()** to select items from the population, is as follows:

```
import random
# Define population and sample size
population = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sample_size = 3
```

```
# Randomly select a sample from the population
sample = random.sample(population, sample_size)
print("Sample:", sample)
```

Tutorial 1.28: To implement **rand()** to select items from the patient registry data, is as follows:

```
import pandas as pd
from IPython.display import display
# Import module to generate random numbers
import random
# Read CSV file and save as dataframe
diabities_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')
# Define the sample size
sample_size = 5
# Get the number of rows in the DataFrame
num_rows = diabities_df.shape[0]
# Generate random indices for selecting rows
random_indices = random.sample(range(num_rows), sample_size)
# Select the rows using the random indices
sample_diabities_df = diabities_df.iloc[random_indices]
display(sample_diabities_df)
```

While random sampling methods like **rand()** help select a representative subset from a broader population, functions such as **train_test_split()** play a pivotal role in organizing this subset into training and testing sets, particularly in supervised learning. By systematically dividing data into dependent and independent variables and ensuring that these splits are both representative and reproducible, **train_test_split()** facilitates the development of models that perform reliably on unseen data.

Tutorial 1.29: To implement **train_test_split()** to select

items from the patient registry data population, is as follows:

```
# Import sklearn train_test_split
from sklearn.model_selection import train_test_split
# Define population and test size
population = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
test_size = 0.2  # Proportion of data to reserve for testing
# Split the population into training and testing sets
train_set, test_set = train_test_split(population, test_size=test_size, random_state=42)
# Display the split
print("Training Set:", train_set)
print("Testing Set:", test_set)
```

Output:

```
Training Set: [6, 1, 8, 3, 10, 5, 4, 7]
Testing Set: [9, 2]
```

Data preparation tasks

Data preparation task are the early steps carried out upon having access to the data. It involves checking the quality of data, cleaning of data, data wrangling, and its manipulation described in detail.

Data quality

Data quality indicates how suitable, accurate, useful, complete, reliable, and consistent the data is for its intended use. Verifying data quality is an important step in analysis and preprocessing.

Tutorial 1.30: To implement checking the data quality of CSV file data frame, is as follows:

- Check missing values with **isna()** or **isnull()**
- Check summary with **describe()** or **info()**
- Check shape with **shape**, size with **size**, and memory usage with **memory_usage()**
- Check duplicates with **duplicated()** and remove duplicate with **drop_duplicates()**

Based on this instruction, let us see the implementation as follows:

```
import pandas as pd
diabities_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')
# Check for missing values using isna() or isnull()
print(diabities_df.isna().sum())
#Describe the dataframe with describe() or info()
print(diabities_df.describe())
# Check for the shape,size and memory usage
print(f'Shape: {diabities_df.shape} Size: {diabities_df.size} Memory Usage: {diabities_df.memory_usage()}')
# Check for the duplicates using duplicated() and drop them if necessary using drop_duplicates()
print(diabities_df.duplicated())
```

Now, we use synthetic transaction narrative data containing unstructured information about the nature of the transaction.

Tutorial 1.30: To implement viewing the text information in the text files (synthetic transaction narrative files), is as follows:

```
import pandas as pd
import numpy as np
# To import glob library for finding files and directories using patterns
import glob
```

```
# To assign the path of the directory containing the text files to a variable
path = "/workspaces/ImplementingStatisticsWithPython/data/chapter1/TransactionNarrative"
# To find all the files in the directory that have a .txt extension and store them in a list
files = glob.glob(path + "/*.txt")
# To loop through each file in the list
for file in files:
    # To open each file in read mode with utf-8 encoding and assign it to a file object
    with open(file, "r", encoding="utf-8") as f:
        print(f.read())
```

Output:

```
Date: 2023-08-01
Merchant: VideoStream Plus
Amount: $9.99
Description: Monthly renewal of VideoStream Plus subscription.
Your subscription to VideoStream Plus has been successfully renewed for $9.99.
```

Tutorial 1.31: To implement checking the data quality of multiple **.txt** files (synthetic transaction narrative files) that contains text information as shown in *Tutorial 1.30* output and to check the quality of information in them, we use **file_size**, **line_count**, **missing_field**, as follows:

```
import os
import glob
def check_file_quality(content):
    # Check for presence of required fields
    required_fields = ['Date:', 'Merchant:', 'Amount:', 'Description:']
```

```
    missing_fields = [field for field in required_fields if fie
ld not in content]
    # Calculate file size
    file_size = len(content.encode('utf-8'))
    # Count lines in the content
    line_count = content.count('\n') + 1
    # Return quality assessment
    quality_assessment = {
        "file_name": file,
        "file_size_bytes": file_size,
        "line_count": line_count,
        "missing_fields": missing_fields
    }
    return quality_assessment
# To assign the path of the directory containing the text
files to a variable
path = "/workspaces/ImplementingStatisticsWithPython
/data/chapter1/TransactionNarrative"
# To find all the files in the directory that have a .txt ext
ension and store them in a list
files = glob.glob(path + "/*.txt")
# To loop through each file in the list
for file in files:
    with open(file, "r", encoding="utf-8") as f:
        content = f.read()
        print(content)
        quality_result = check_file_quality(content)
        print(f"\nQuality Assessment for {quality_result['fil
e_name']}:")
        print(f"File Size: {quality_result['file_size_bytes']} b
ytes")
```

```
        print(f"Line Count: {quality_result['line_count']} lines")
        if quality_result['missing_fields']:
            print("Missing Fields:", ', '.join(quality_result['missing_fields']))
        else:
            print("All required fields present.")
        print("=" * 40)
```

Output (Only one transaction narrative output is shown):

```
Date: 2023-08-01
Merchant: VideoStream Plus
Amount: $9.99
Description: Monthly renewal of VideoStream Plus subscription.

Your subscription to VideoStream Plus has been successfully renewed for $9.99.


Quality Assessment for /workspaces/ImplementingStatisticsWithPython/data/chapter1/TransactionNarrative/3.txt:
File Size: 201 bytes
Line Count: 7 lines
All required fields present.
========================================
```

Cleaning

Data cleansing involves identifying and resolving inconsistencies and errors in raw data sets to improve data quality. High-quality data is critical to gaining accurate and

meaningful insights. Data cleansing also include data handling. Different ways for data cleaning or data handling are described below.

Missing values

Missing values refer to data points or observations with incomplete or absent information. For example, in a survey, if people do not answer a certain question, the related entries will be empty. Appropriate methods, like imputation or exclusion, are used to address them. If there are missing values then one way is to drop missing value as shown in *Tutorial 1.32.*

Tutorial 1.32: To implement finding the missing value and dropping them.

Let us check **prize_csv_df** data frame for null values and drop the null ones, as follows:

```
import pandas as pd
from IPython.display import display
# Read the prize csv file from the direcotory
prize_csv_df = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.csv")
# Display the dataframe null values count
print(prize_csv_df.isna().sum())
```

Output:

```
year                  374
category              374
overallMotivation     980
laureates__id          49
laureates__firstname   50
laureates__surname     82
laureates__motivation  49
laureates__share       49
```

Since **prize_csv_df** have null values, let us drop them and view the count of null values after drop as follows:

```
print("\n \n **** After droping the null values in prize_csv_df****")
after_droping_null_prize_df = prize_csv_df.dropna()
print(after_droping_null_prize_df.isna().sum())
```

Finally, after applying the above code, the output will be as follows:

```
**** After droping the null values in prize_csv_df****
year                    0
category                0
overallMotivation       0
laureates__id           0
laureates__firstname    0
laureates__surname      0
laureates__motivation   0
laureates__share        0
dtype: int64
```

This shows there are now zero null values in all the column.

Imputation

Imputation means to place a substitute value in place of the missing values. Like constant value imputation, mean imputation, mode imputation.

Tutorial 1.33: To implement imputing the mean value of the column **laureates__share**.

Mean imputation only imputes the mean value of numeric data types as **fillna()** expects scalar, so we cannot use the **mean()** method to fill missing values in object columns.

```
import pandas as pd
from IPython.display import display
# Read the prize csv file from the direcotory
```

```
prize_csv_df = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.csv")
# View the number of null values in original DataFrame
print("Null Value Before",prize_csv_df['laureates__share'].isna().sum())
# Calculate the mean of each column
prize_col_mean = prize_csv_df['laureates__share'].mean()
# Fill missing values with column mean,  inplace = True will replace the original DataFrame
prize_csv_df['laureates__share'].fillna(value=prize_col_mean, inplace=True)
# View the number of null values in the new DataFrame
print("Null Value After",prize_csv_df['laureates__share'].isna().sum())
```

Output:

```
Null Value Before 49
Null Value After 0
```

Also to fill missing values in object columns, you have to use a different strategy, such as using a constant value i.e, **df[column_name].fillna(' ')**, a mode value, or a custom function..

Tutorial 1.34: To implement imputing the mode value in the object data type column.

```
import pandas as pd
from IPython.display import display
# Read the prize csv file from the direcotory
prize_csv_df = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.csv")
# Display the original DataFrame null values in object data type columns
print(prize_csv_df.isna().sum())
```

```
# Select the object columns
object_cols = prize_csv_df.select_dtypes(include='object').columns
# Calculate the mode of each object data type column
col_mode = prize_csv_df[object_cols].mode().iloc[0]
# Fill missing values with the mode of each object data type column
prize_csv_df[object_cols] = prize_csv_df[object_cols].fillna(col_mode)
# Display the DataFrame column after filling null values in object data type columns
print(prize_csv_df.isna().sum())
```

Output:

```
year                  374
category              374
overallMotivation     980
laureates__id          49
laureates__firstname   50
laureates__surname     82
laureates__motivation  49
laureates__share       49
dtype: int64
year                  374
category                0
overallMotivation       0
laureates__id          49
laureates__firstname    0
laureates__surname      0
laureates__motivation   0
laureates__share       49
dtype: int64
```

Duplicates

Data may be duplicated or contains duplicate value. The duplicacy will affect the final statistical result. Hence, to prevent duplicacy, identifying and removing duplicates is necessary step as explained in this section. Best way to handle duplicate is to identify and remove duplicates.

Tutorial 1.35: To implement identifying and removing duplicate rows in data frame with **duplicated()**, as follows:

```
# Identify duplicate rows and display their index
print(prize_csv_df.duplicated().index[prize_csv_df.duplicated()])
```

Since, there is no duplicate the output is empty it displays indexes of duplicates as follows:

```
Index([], dtype='int64')
```

Also, you can find the duplicate values in a specific column by using the following code:

```
prize_csv_df.duplicated(subset=['name_of_the_column'])
```

To remove duplicates, **drop()** method can be used, syntax will be **dataframe.drop(labels, axis='columns', inplace=False)**. Drop can be applied to row and index using label and index values as follows:

```
import pandas as pd
# Create a sample dataframe
people_df = pd.DataFrame({'name': ['Alice', 'Bob', 'Charlie'], 'age': [25, 30, 35], 'gender': ['F', 'M', 'M']})
# Print the original dataframe
print("original dataframe \n",people_df)
# Drop the 'gender' column and return a new dataframe
new_df = people_df.drop('gender', axis='columns')
# Print the new dataframe
print("dataframe after drop \n",new_df)
```

Output:

```
original dataframe
      name  age gender
0    Alice   25      F
1      Bob   30      M
2  Charlie   35      M
dataframe after drop
      name  age
0    Alice   25
1      Bob   30
2  Charlie   35
```

Outliers

Outliers are data points that are very different from the other data points. They can be much higher or lower than the standard range of values. For example, if the heights of ten people in centimeters are measured, the values might be as follows:

160, 165, 170, 175, 180, 185, 190, 195, 200, 1500.

Most of the heights are alike but the last measurement is much larger than the others. This data point is an outlier because it is not like the rest of the data. The best way to handle outliers is to identify outliers and then correct, resolve, or leave as needed. Ways to identify outliers are to compute mean, standard deviation, and quantile (a common approach is to compute interquartile range). Another way to identify outliers is by computing the z-score of the data points and then considering points beyond the threshold values as outliers.

Tutorial 1.36: To implement identifying outliers in a data frame with zscore.

Z-score measures how many standard deviations a value is from the mean. In the following code, **z_score** identifies

outliers in the laureates' share column:

```
import pandas as pd
import numpy as np
# Read the prize csv file from the direcotory
prize_csv_df = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.csv")
# Calculate mean, standard deviation and Z-scores for the column
z_scores = np.abs((prize_csv_df['laureates__share'] - prize_csv_df['laureates__share'].mean()) / prize_csv_df['laureates__share'].std())
# Define a threshold for outliers (e.g., 4)
threshold = 2
# Display the row index of the outliers
print(prize_csv_df.index[z_scores > threshold])
```

Output:

```
Index([ 17,  18,  22,  23,  34,  35,  48,  49,  54,  55,
  62,  63,
       73,  74,  86,  87,  97,  98, 111, 112, 144, 145,
 146, 147,
      168, 169, 180, 181, 183, 184, 215, 216, 242, 243,
 249, 250,
      255, 256, 277, 278, 302, 303, 393, 394, 425, 426,
 467, 468,
      471, 472, 474, 475, 501, 502, 514, 515, 556, 557,
 563, 564,
      607, 608, 635, 636, 645, 646, 683, 684, 760, 761,
```

```
      764,  765,
7.        1022, 1023],
8.       dtype='int64')
```

The output shows the row index of the outliers in the laureates' share column of the **prize.csv** file. Outliers are values that are unusually high or low compared to the rest of the data. The code uses a z-score to measure how many standard deviations a value is from the mean of the column. A higher z-score means a more extreme value. The code defines a threshold of two, which means that any value with a z-score greater than two is considered an outlier.

Additionally, preparing data, cleaning it, manipulating it, and doing data wrangling includes the following:

- Cheking typos and spelling errors. Python provides libraries like **PySpellChecker**, **NLTK**, **TextBlob**, or **Enchant** to check typos and spelling errors.
- Data transformation is a change from one form to another desired form. It involves aggeration, conversion, normalization, and many more, they are covered in detail in *Chapter 2*, *Exploratory Data Analysis*.
- Handling inconsistencies which involve identifying conflicting information and resolving them. For example, the body temperature is listed as 1400 Celsius which is not correct.
- Standardize format and units of measurements to ensure consistency.
- Further data integrity and validation ensures that data is unchanged, not altered or corrupted. Data validation verifies that the data to be used is correct (use techniques like validation rules, manual review).

Wrangling and manipulation

It means making raw data usable through cleaning, transformation, or other ways. It involves cleaning, organizing, merging, filtering, sorting, aggregating, and reshaping data. Helping you analyze, organize, and improve your data for informed insights and decisions.

The various useful data wrangling and manipulation method in Python are as following:

- **Cleaning**: Some of the methods used to clean the data, along with their syntax, are as follows:

  df.dropna(): Removing missing values.

  df.fillna(): Filling missing values.

  df.replace(): Replacing values.

  df.drop_duplicates(): Removing duplicate.

  df.drop(): Removing specific rows or columns.

  df.rename(): Renaming columns.

  df.astype(): Changing data types.

- **Transformation**: Some of the methods used for data transformation, together with their syntax, are as follows:

  df.apply(): Applying a function.

  df.groupby(): Grouping data.

  df.pivot_table(): Creating pivot tables to summarize.

  df.melt(): Unpivoting or melting data.

  df.sort_values(): Sorting rows.

  df.join(), df.merge(): Combining data.

- **Aggregation**: Some methods used for data aggregation, together with their syntax, are as follows:

  df.groupby().agg(): Aggregate data using specified functions.

  df.groupby().size(), df.groupby().count(), df.groupby().mean(): Calculating common aggregation metrics.

- **Reshape**: Some methods used for data reshape,

together with their syntax, are as follows:

```
df.transpose(): Transposing rows and columns.
df.stack(), df.unstack(): Stacking and unstacking.
```

- **Filtering and subset selection**: Some methods for data filtering and subset selection are as follows:

```
df.loc[], df.iloc[]: Selecting subsets.
df.query(): Filtering data using a query.
df.isin(): Checking for values in a DataFrame.
df.nlargest(), df.nsmallest(): Selecting the largest or s
mallest values.
```

- **Sorting**: Some methods used for sorting are as follows:

```
df.sort_values(): Sorts a DataFrame by one or more col
umns.
Ascending or descending order.
df.sort_index(): Sorts a DataFrame based on the row in
dex.
sort(): Sorts lists in ascending and descending order
```

- **String manipulation**: Some methods used for string manipulation are as follows:

```
str.strip(), str.lower(), str.upper(), str.replace()
```

Moreover, adding new columns, variables, statistical modeling, testing and probability distribution, and exploratory data analysis is also part of data wrangling and manipulation, which will be covered in *Chapter 2, Exploratory Data Analysis*.

Conclusion

Statistics provides a structured framework for understanding and interpreting the world around us. It empowers us to gather, organize, analyze, and interpret information, thereby revealing patterns, testing hypotheses, and informing decisions. In this chapter, we

examined the foundations of data and statistics: from the distinction between qualitative (descriptive) and quantitative (numeric) data to the varying levels of measurement—nominal, ordinal, interval, and ratio. We also considered the scope of analysis in terms of the number of variables involved—whether univariate, bivariate, or multivariate—and recognized that data can originate from diverse sources, including surveys, experiments, and observations.

We explored how careful data collection methods—whether sampling from a larger population or studying an entire group—can significantly affect the quality and applicability of our findings. Ensuring data quality is key, as the validity and reliability of statistical results depend on accurate, complete, and consistent information. Data cleaning addresses errors and inconsistencies, while data wrangling and manipulation techniques help us prepare data for meaningful analysis.

By applying these foundational concepts, we establish a platform for more advanced techniques. In the upcoming Chapter 2, Exploratory data analysis we learn to transform and visualize data in ways that reveal underlying structures, guide analytical decisions, and communicate insights effectively, enabling us to extract even greater value from data.

---

1 Source: **https://scikit-learn.org/stable/datasets/toy_dataset.html#iris-dataset**

2 Source: **https://github.com/jdorfman/awesome-json-datasets#nobel-prize**

3 Source: **https://github.com/jdorfman/awesome-json-datasets#nobel-prize**

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com